Hashing and Hash Functions

Why Hashing?

(key, value) pairs

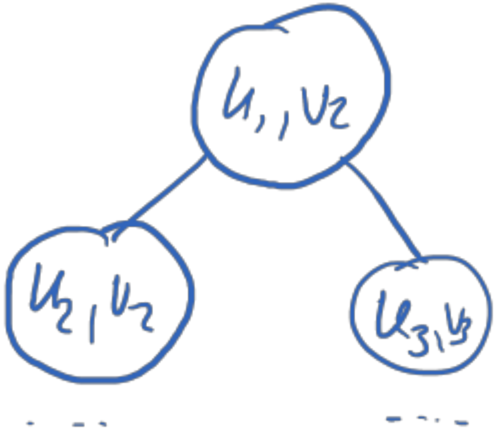
(Student ids) —> (Student record)

Goal: efficiently insert, look up, and delete pairs

array, linked list $O(N)$

(k_1, v_1)	(k_2, v_2)	....

Tree-type $O(\log N)$



$N =$ # key, value pairs stored

Better

Hashing: The Main Idea

$f(key) \rightarrow index$

index	0	1	2	3	4	. . .
content			(k, v)			. . .

hash functions:

key → $\boxed{}$ → location

(large range) (smaller range)

location: valid hash index

≥ 0, $<$ array length

Ideal Hashing

In this case, one record takes one index !

Assume:

- We want to store **100 student records**
- We use an **array of size 100**
- We use **student IDs as keys**

student IDs: 11 000, 11 001, 11 002, ..., 11 099

hash function:

```
int hash(int key) {
    return key - 11000;
}   //or: return key % 11000;
```

Perfect Hash Function

A perfect hash function maps every key to a **unique** hash index.

Operations

K: key type
V: value type

field: V[ ] array

```
void insert(K key, V value) {
    array[hash(key)] = value;
}

V lookup(K key) {
    return array[hash(key)];
}

void remove(K key) {
    array[hash(key)] = null;
}
```

For Ideal Hashing

Real World: UW Student IDs

Not always perfect, collisions happen a lot

10 digit numbers: 9021453190
9033879101
9024357190

collision

Can we find a perfect hash function?

max. Java array length
~2 billion

Last 3 digits?

Properties of a “Good” Hash Function

1. must be deterministic
2. should achieve uniform distribution across output range
3. should minimize collisions
4. should be fast and easy to compute

Java API for Hash Functions

`int hashCode()` method:

- instance method of **Object** type
- Java has implementations for built-in data types (String, Double, etc.)
- we can override it for our own data types

Steps for using `int hashCode()`:

1. call `hashCode()` on key object
2. convert result (in range of integer) to valid hash index (abs, modulo)

Math.abs(key.hashCode()) % array.length

↳ valid hash index

Hash Tables and Collision Handling

Hash Table Properties

- hash table: array that contains (key, value) pairs
- table size: current capacity (array length)
- load factor (LF): $\dfrac{(number\ of\ key,value\ pairs\ in\ table)}{(table\ size)}$

Table Size and Collisions

- Experiment repeated 10 times per table size:
  - 100 random integer keys
  - “hash function”: abs(key) % (table size)

How often do collisions occur for different table sizes?

# keys	table size	load factor	# of collisions
100	10,000	0.01	0 or 1
100	1,000	0.1	3-7
100	100	1	35-47
100	10	10	90

.7-.8

Resizing: When?

- When the table is "full":

$$\text{load\_factor} \geq \text{load\_factor\_threshold}$$

- Who defines the threshold?

Let user of our data structure set it, with good default (.7-.8).

Resizing: Rehashing

- hash function: key % (table size)
- LF threshold: 0.7

0	1	2	3	4
30		17	88	

Currently the LF = 0.6, however if adding a new one, it will exceed the threshold

double table size

0	1	2	3	4	5	6	7	8	9
30		17	88						

rehashing

0	1	2	3	4	5	6	7	8	9
30							17	88	

Because the table size doubled, so the hash function was also modified

Resizing: Complexity

- Resizing: O(1)
- Rehashing: O(N), where N is the number of keys in the old hash table

- Amortized over many inserts: O(1)

☆ Although rehashing is expensive, it happens rarely, it is spread to many inserts

Collision Handling: Open Addressing

- Each element of hash table stores at most one key, value pair
- If there is a collision, look for next “open address”

Open Addressing: Linear Probing

Problem: Primary clustering → consecutive cells get filled and cause long probe chains, resulting very poor performance

- Probe sequence: $H_K, H_K + 1, H_K + 2, H_K + 3, \ldots$
- $H_k = hash\ index$

key	seq
166	1
359	7, 8
263	10, 11(0), 1, 2, 3, 4, 5

hash function: key % (table size)

0	1	2	3	4	5	6	7	8	9	10
440	166	266	124	246	263		337	359		351

Open Addressing: Look Up

(If we encounter any empty spot, it means that key-value pair was never inserted)

- Look Up: 263, 330

$H_K = 0$

$H_K = 10$

0	1	2	3	4	5	6	7	8	9	10
440	166	266	124	246	263		337	359		351

- Delete 266, Look Up: 263

$H_K = 10$

0	1	2	3	4	5	6	7	8	9	10
440	166	DEL	124	246	263		337	359		351

As a result, we need to leave the field as DEL when deleting any pair. Otherwise it will affect the look-up of others

Problem: Can get stuck in a cycle, since it does not guarantee it will try every index in the table

Open Addressing: Quadratic Probing

- Add polynomials of order 2: $H_K + c_0 i + c_1 i^2$ (usually: $c_0 = 0, c_1 = 1$)
- i: step number
- Probe sequence: $H_K, H_K + 1^2, H_K + 2^2, H_K + 3^2, \ldots$

key	seq
166	1
359	7, $7+1^2=8$
263	10, $10+1^2=11$ (0), $10+2^2=14$ (3), $10+3^2=19$ (8), 4, 2, 2, 4, 8, 3, 0, 10, 0, 3, ...

hash function: key % (table size)

0	1	2	3	4	5	6	7	8	9	10
440	166	266	124	246			337	359		351

Open Addressing: Double Hashing

- Second hash function for step size (s)

$$s = \boxed{hash2}(key)$$

→ 2nd indep. hash function

- Probing sequence: $H_K, H_K + s * 1, H_K + s * 2, H_K + s * 3, \ldots$

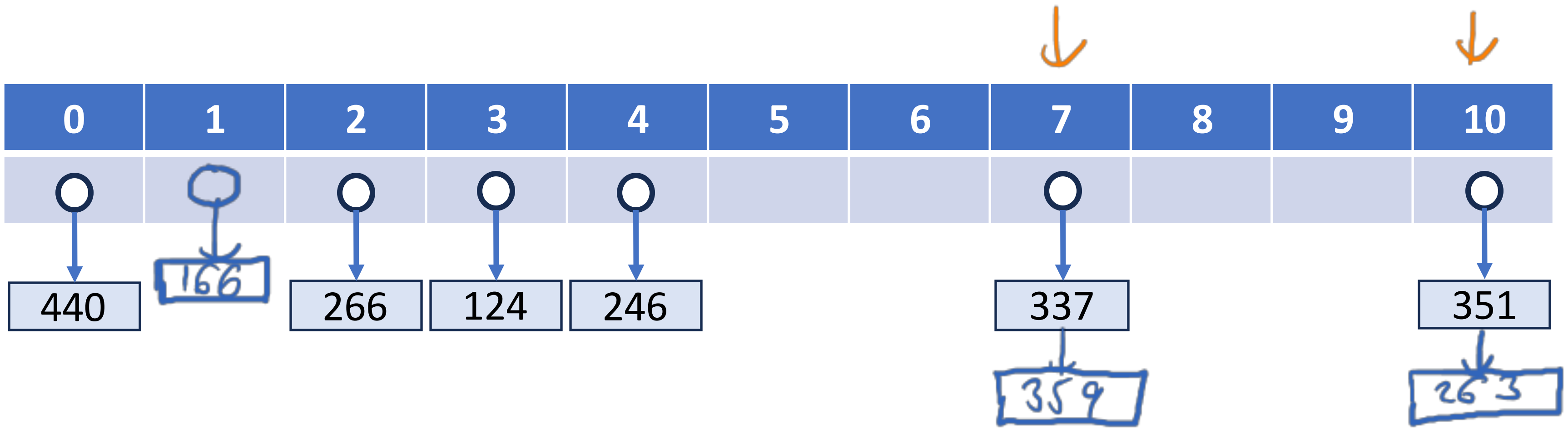
Collision Handling: Chaining

Each element of the hash table is a "chain" that can hold multiple (key, value) pairs.

linked list

key	seq
166	1
359	7
263	10

hash function: key % (table size)



Complexities

$N \hat{=} \#$ key, value pairs in table

	worst case	average case	best case
insert (put)	$O(N)$	$O(1)$	$O(1)$
look up (get)	$O(N)$	$O(1)$	$O(1)$
remove	$O(N)$	$O(1)$	$O(1)$